A Group Lock Algorithm, With Applications

by

Isaac Dimitrovsky

Ultracomputer Note #112

November, 1986

[redacted]

A Group Lock Algorithm, With Applications

by

Isaac Dimitrovsky

Ultracomputer Note #112
November, 1986

ABSTRACT

In this note I introduce the concept of a group lock. This is a generalization of P and V that can be used in writing asynchronous parallel algorithms. I then present an algorithm for group locking that has a nice time complexity property. This algorithm is used to define several fast and simple parallel algorithms. Among these are algorithms for a parallel queue, a parallel stack, atomic read/writes on multiword records, a parallel heap, and a parallel dictionary.

1. Introduction

In this note I introduce the concept of a group lock. This is a generalization of P and V that can be used in writing asynchronous parallel algorithms. I then present an algorithm for group locking that has a nice time complexity property. This algorithm is used to define several fast and simple parallel algorithms. Among these are algorithms for a parallel queue, a parallel stack, atomic read/writes on multiword records, a parallel heap, and a parallel dictionary.

The algorithm is defined for the paracomputer model of computation [GGK]. In this model, P independent processors share a common memory. Concurrent reads and writes are allowed to locations in the shared memory. In addition an operation faa(addr,incr) is provided that returns the current value in the word pointed to by addr, while as a side effect adding incr to that value (faa stands for fetch and add). If several reads, writes, and faa's are simultaneously addressed to the same location, they all complete in one cycle and the result is guaranteed to be the same as if the operations had been done one after the other in some unspecified order. Aside from this, each processor has a conventional instruction set. See [GGK] for an explanation of how this model can be closely approximated by an actual machine (the NYU Ultracomputer).

The group lock problem is as follows. Define a data structure glocktype, and three routines glock(glocktype), gsync(glocktype), and gunlock(glocktype) such that if we call the code segment:

```
glock(g);
{operation X}
gunlock(g);
```

from any number of processors and in any pattern, these calls will be separated into groups that are executed one group at a time. More precisely, no matter what the pattern of the calls to the above code segment, the calls will be executed so that the following requirements are fulfilled:

(1) Each call can be placed in a unique group of calls. The execution of {operation X} from calls in two different groups does not overlap, i.e. all the calls in one group will finish executing {operation X} before any of the calls in the other group start.

(2) All the calls in a group start executing {operation X} within constant time of each other.

(3) If all the calls in a group have finished executing {operation X} and some other calls being executed have not yet started {operation X}, then within constant time another group of calls will start executing {operation X}.

(4) If some of the calls in a group execute gsync(g) from within {operation X}, then they will stop and wait for all the remaining calls in the group to execute gsync(g) as well. If all the calls in the group execute gsync(g), then within constant time afterward all the calls will proceed. Thus, gsync(g) provides barrier synchronization among all the calls in each group.

The operations glock and gunlock can be seen as a looser form of P and V, with {operation X} replacing the critical section (note that P and V may be used as glock and gunlock, respectively; this would be a valid, though inefficient, group lock). The {operation X} may be performed by a whole group of processes at once, as opposed to P and V where only one process at a time can perform the critical section. However, the execution of {operation X} by each group of processes is mutually exclusive in the same way that the execution of the critical section by each process is mutually exclusive.

The algorithm I present has the additional property that if {operation X} always completes in time O(f), then the locked operation:

```
glock(g);
{operation X}
gunlock(g);
```

also will always complete in time O(f). In other words, no call will have to wait for more than O(1) groups to execute {operation X} before being taken into a group and executing {operation X} itself.

What can we use this group lock for? Suppose we know a synchronous O(f) algorithm to do k operations in parallel with k processors for any given k. Using a group lock, we can extend the algorithm to be able to handle asynchronous requests for operations coming in from any number of processors in any pattern. Moreover, each request for an operation is still guaranteed to terminate in time O(f), no matter how many other requests are active.

For example, consider the parallel stack problem. We extend the definition of LIFO order to parallel stacks in a reasonably obvious way, to wit: if we start pushing item B after we finish pushing item A, and we finish pushing item B before we start popping item A, then we must start popping item B before we finish popping item A. We know how to do k stack operations synchronously with k processors (use faa on an index into an array of items, doing the inserts first followed by the deletes). Using a group lock, we can now define a simple parallel stack structure that can handle general patterns of push and pop requests, with each request guaranteed to complete in constant time (see the parallel stack section below for a more detailed description).

2. The Group Lock Algorithm

Within the group lock algorithm, I use several routines that are defined and fully explained in [GLR]. For completeness, the code for these routines is given below.

```
const
        maxpes = {> maximum number of processors executing at once};

type
        rwlocktype = record {readers/writers lock with writer priority}
                wflag:integer; {initially 0}
                sem:integer; {initially maxpes}
        end;

function tir(var i:integer; delta,bound:integer):boolean;
begin
        tir := false;
        if i+delta <= bound then begin
                if faa(i,delta)+delta <= bound then
                        tir := true
                else
                        faa(i,-delta);
        end;
end;

function tdr(var i:integer; delta:integer):boolean;
begin
        tdr := false;
        if i >= delta then begin
                if faa(i,-delta) >= delta then
                        tdr := true
                else
                        faa(i,delta);
        end;
end;
```

```
procedure pc(var sem:integer; delta:integer);
begin
        waitfor(tdr(sem,delta));
end;

procedure vc(var sem:integer; delta:integer);
begin
        faa(sem,delta);
end;

procedure p(var sem:integer);
begin
        pc(sem,1);
end;

procedure v(var sem:integer);
begin
        vc(sem,1);
end;

procedure rwrlock(var rw:rwlocktype);
begin
        waitfor(rw.wflag = 0);
        pc(rw.sem,1);
end;

procedure rwrunlock(var rw:rwlocktype);
begin
        vc(rw.sem,1);
end;

procedure rwwlock(var rw:rwlocktype);
begin
        faa(rw.wflag,1);
        pc(rw.sem,maxpes);
end;

procedure rwwunlock(var rw:rwlocktype);
begin
        vc(rw.sem,maxpes);
        faa(rw.wflag,-1);
end;
```

The code for the group lock algorithm follows:

```
type
        glocktype = record
                cycle,active,locked:integer; {initially 0}
                waitcount:array[0..1] of integer; {initially 0}
                lock:rwlocktype;
                nprocessors,syncvar:integer;
        end;

procedure glock(var g:glocktype);
var mycycle:integer; first:boolean;
begin
        mycycle := g.cycle;
        faa(g.waitcount[mycycle],1);
        rwrlock(g.lock);
        first := (faa(g.active,1)=0);
        rwrunlock(g.lock);
        faa(g.waitcount[mycycle],-1);
        if first then begin
                waitfor(g.waitcount[1-g.cycle] = 0);
                g.cycle := 1-g.cycle;
                rwwlock(g.lock);
                g.nprocessors := g.active;
                g.syncvar := 0;
                g.locked := 1;
        end;
        waitfor(g.locked = 1);
end;

procedure gunlock(var g:glocktype);
begin
        if faa(g.active,-1) = 1 then begin
                g.locked := 0;
                rwwunlock(g.lock);
        end;
end;

procedure gsync(var g:glocktype);
var wasless:boolean;
begin
        wasless := (g.syncvar<g.nprocessors);
        if faa(g.syncvar,1) = 2*g.nprocessors-1 then
                g.syncvar := 0;
        waitfor(wasless <> (g.syncvar<g.nprocessors));
end;
```

See the last section in this note for a long and messy proof that this algorithm has the properties stated in the introduction.

3. Some Applications

In this section, I give some simple examples of how the group lock may be used in writing asynchronous parallel algorithms.

3.1. A Parallel Queue

The group lock algorithm above may be used to define a queue algorithm that is simpler and more space-efficient than the one defined in [GLR]. Specifically, we can separate the queue operations into groups and do all the inserts in each group followed by all the deletes. We then need not worry about cell contention and so the complete code for a parallel queue simplifies to:

```
type
        itemtype = {type of items in the queue};
        queuetype = record
                i,d,nitems,size : integer; {i,d,nitems are initially 0}
                items : array[0..size-1] of itemtype;
                g : glocktype;
        end;

function insert(item:itemtype; var q:queuetype):boolean;
var myi : integer;
begin
        glock(q.g);
        if (insert := tir(q.nitems,1,q.size)) then begin
                myi := faa(q.i,1) mod q.size;
                q.items[myi] := item;
                if myi = q.size-1 then
                        faa(q.i,-q.size);
        end;
        gsync(q.g);
        gunlock(q.g);
end;

function delete(var item:itemtype; var q:queuetype):boolean;
var myd : integer;
begin
        glock(q.g);
        gsync(q.g);
        if (delete := tdr(q.nitems,1)) then begin
                myd := faa(q.d,1) mod q.size;
                item := q.items[myd];
                if myd = q.size-1 then
                        faa(q.d,-q.size);
        end;
        gunlock(q.g);
end;
```

Because of the time complexity property of the group lock algorithm, inserts and deletes are guaranteed to complete in constant time no matter how many other inserts and deletes are active. (It should be clear that the code segments between the glocks and gunlocks take constant time).

Because this algorithm precludes cell contention, we do not need any of the methods suggested in [GLR] to avoid cell contention. Thus, we save the cell vacant flags, semaphores, etc. that these methods require in each place in the queue, and the algorithm is simplified as well.

3.2. A Parallel Stack

The group lock algorithm above may be used to define a simple and efficient parallel stack algorithm. The code for the algorithm follows:

```
type
        itemtype = {type of items in the stack};
        stacktype = record
                i,nitems,size : integer; {i,nitems are initially 0}
                items : array[0..size-1] of itemtype;
                g : glocktype;
        end;

function push(item:itemtype; var s:stacktype):boolean;
var myi : integer;
begin
        glock(s.g);
        if (push := tir(s.nitems,1,s.size)) then
                s.items[faa(s.i,1)] := item;
        gsync(s.g);
        gunlock(s.g);
end;

function pop(var item:itemtype; var s:stacktype):boolean;
var myd : integer;
begin
        glock(s.g);
        gsync(s.g);
        if (pop := tdr(s.nitems,1)) then
                item = s.items[faa(s.i,-1) - 1];
        gunlock(s.g);
end;
```

Because of the time complexity property of the group lock algorithm, pushes and pops are guaranteed to complete in constant time no matter how many other pushes and pops are active. (It should be clear that the code segments between the glocks and gunlocks take constant time).

3.3. Atomic Read/Writes on Multiword Records

The following algorithm is due to Malcolm Harrison. The group lock algorithm above may be used to define a simple and efficient algorithm for a parallel database-type application in which we must read and write multiword records in parallel, and these reads and writes must be atomic.

A conventional approach to this would be to place a read/write lock with writer (resp. reader) priority on each record. This can lead to serialization of many writers to the same record, and starvation of a reader from (resp. writer to) a record by a stream of writers to (resp. readers from) the same record. The algorithm below avoids these problems. In fact, all reads and writes are guaranteed to complete in constant time. The code for the algorithm follows:

```
type
        recordtype = {big messy record that must be read/written atomically};
        databasetype = record
                recs : array[0..size-1] of record
                        rec : recordtype;
                        wlock : integer; {initially 1}
                end;
                g : glocktype;
        end;

procedure rwrite(rec:recordtype; n:integer; var d:databasetype);
begin
        glock(d.g);
        if tdr(d.recs[n].wlock,1) then begin
                d.recs[n].rec := rec;
                faa(d.recs[n].wlock,1);
        end;
        gsync(d.g);
        gunlock(d.g);
end;

procedure rread(var rec:recordtype; n:integer; var d:databasetype);
begin
        glock(d.g);
        gsync(d.g);
        rec := d.recs[n].rec;
        gunlock(d.g);
end;
```

Because of the time complexity property of the group lock algorithm, rreads and rwrites are guaranteed to complete in constant time no matter how many other rreads and rwrites are active. (It should be clear that the code segments between the glocks and gunlocks take constant time).

3.4. A Parallel Heap

The group lock algorithm above can be used to define an efficient parallel heap. The reference [PVW] contains several parallel algorithms for 2-3 trees. They consider a case where we are given a 2-3 tree T with n leaves, k synchronized processors P_1 ... P_k, and k items a_1 ... a_k, with processor i knowing item i. They provide O(log(n)+log(k)) algorithms for, among other things, inserting a_1 ... a_k into the tree and deleting a_1 ... a_k from the tree.

These algorithms may be used together with a group lock to define a parallel heap. This heap will support inserts of arbitrary items and deletion of the minimum item in time O(log(n)+log(P)), where n is the number of items in the heap and P is the number of processors.

We divide the heap operations into groups, and within each group do the heap inserts followed by the delete minimums. For heap inserts, we want to do exactly what the insert algorithm in [PVW] tells us how to do. For delete minimums, we first have a phase where the k processors that want to delete a minimum go down the 2-3 tree in parallel and find the k smallest items. On a paracomputer, this can easily be done in log(n) time (we can maintain in each node a count of its leaf descendants). Then, we can use the delete algorithm in [PVW] to delete these items.

3.5. A Parallel Dictionary

The algorithms in [PVW] can also be used to define a parallel dictionary. This dictionary will support inserts, searches, and deletes of arbitrary items in time O(log(n)+log(P)), where n is the number of items in the dictionary and P is the number of processors. We divide the dictionary operations into groups, and within each group do the inserts, followed by the searches, followed by the deletes.

3.6. A Starvation-free Semaphore

The procedures p and v given in [GLR] and reproduced above suffer from the potential problem of starvation. Jim Lipkis suggested using the following as a simple starvation-free semaphore:

```
type
        starvefreesemtype = record
                s : integer {initially 1};
                g : glocktype;
        end;

procedure starvefreep(var sem:starvefreesemtype);
begin
        glock(sem.g);
        p(sem.s);
end;

procedure starvefreev(var sem:starvefreesemtype);
begin
        v(sem.s);
        gunlock(sem.g);
end;
```

That this is starvation-free does not follow from the group lock properties given above (in fact, it is not true of all group locks). However, an examination of the proof below shows that when we use the group lock algorithm given above we do get a starvation-free semaphore. This follows from the fact that each starvefreep operation, upon executing glock, insures that it will be taken into one of the next two groups of critical sections.

4. Proof of the Group Lock Algorithm

In this section I outline a proof that the group lock algorithm given above has the properties given in the introduction. The proof proceeds by showing that when any number of processors call the code segment:

```
glock(g);
{operation X}
gunlock(g);
```

in any pattern over time, we always cycle through the three states described below.

Note that initially there are no previous groups of calls, g.active=0, g.locked=0, and g.lock is completely unlocked, so we have a particular case of state 1. In all the states, if there are any calls from previous groups still being executed, they have finished accessing the glocktype structure.

state 1

g.active=0, g.locked=0, g.lock is not write-locked. If there are any calls being executed they are all before the assignment to first in glock. If there are any calls being executed, then within constant time we will go into state 2.

state 2

g.active>0, g.locked=0, g.lock is not write-locked. One of the calls executing the assignment to first will set first to true. All other calls coming in will set first to false. Within constant time, all calls coming in but the first that pass the assignment to first will reach the waitfor loop at the end of glock. However, they will not pass this loop until the first call sets g.locked to 1. The first call will reach the first waitfor loop in glock in constant time. Now, g.cycle is only changed by the first call in each group in the statement after this waitfor loop, so it will not be changed while the first call is in the loop. Therefore, the waitfor loop is only waiting for calls that are already being executed to pass through the first part of glock, i.e. it is not waiting for any new calls that may be coming in. It follows that the waitfor loop only takes constant time. Because g.lock is a readers-writers lock with writer priority and the code between the rwrlock and rwrunlock takes constant time, the rwwlock will also only take constant time. Once g.lock is write-locked, all calls that have not yet reached the assignment to first in glock will be held up at the rwrlock before this assignment. We take as the current group all the calls that have already done this assignment. Immediately after rwwlock is executed g.active will be the number of calls in the current group, so we set g.nprocessors to this value. We then set g.locked to 1. Within constant time, therefore, we go from state 2 into state 3.

state 3

g.nprocessors = the number of calls in the current group, g.active>0, g.locked=1, g.lock is write-locked. All calls in the current group will proceed with {operation X}, within constant time of each other. Within {operation X} we can call gsync, which is just a standard barrier synchronization algorithm. Any new calls coming in are held up at the rwrlock before the assignment to first in glock. As each call completes {operation X} and executes gunlock, g.active is decremented. The "last" call to complete resets g.active to 0, and then sets g.locked to 0 and releases the write lock on g.lock. Thus, we return to state 1.

It follows from the above that the algorithm fulfills the group lock properties in the introduction. In addition, it fulfills the time complexity property. This is because of the waitfor loop that the first call in each group executes in glock. Each call increments g.waitcount[0] or g.waitcount[1] at the start of glock, and decrements it only after it passes the assignment to first and is therefore guaranteed to be taken into the current group. It follows that each call will miss at most two groups of calls before it is waited for and taken into a group.

5. References

[GGK]

A. Gottlieb, R. Grishman, C. P. Kruskal, et. al. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, in IEEE Transactions on Computers, Vol. C-32 No. 2, Feb. 1983, pp. 175-189.

[GLR]

A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors, in ACM Transactions on Programming Languages and Systems, Vol. 5 No. 2, April 1983, pp. 164-189.

[PVW]

W. Paul, U. Vishkin, and H. Wagener. Parallel Computation on 2-3 Trees, New York University Technical Report #70/Ultracomputer note #52, April 1983.

NYU UCN-112 c.1
Dimitrovsky, Isaac
A group lock algorithm,
with applications.